Implementación TDA — Lista, Pila y Cola

[7541/9515] Algoritmos y Programación II

Segundo cuatrimestre de 2021

Alumno:	BENITEZ, Nahuel Tomas
Número de padrón:	106841
Email:	ntbenitez@fi.uba.ar

1. Introducción Teórica

Un Tipo de Dato Abstracto (TDA) es un tipo de dato que se define a partir de funciones, valores, bibliotecas... Tienen un conjunto mínimo de operaciones esenciales y son muy útiles. Algunos ejemplos de TDA son las listas y sus derivados, que se basan en la idea de los nodos enlazados. Esto es que se pueden crear sucesivos nodos conectados entre sí a través de punteros. Una de las ventajas de usar nodos es que al usar memoria dinámica, no hay límite de reserva de memoria(más allá de la capacidad intrínseca de la misma), ya que no es necesario que esta esté contigua, como sí lo es por ejemplo para vectores. Hay distintas implementaciones de listas, por ejemplo:

LISTA

Simplemente Enlazada: Consiste en que cada nodo solo conoce a su nodo siguiente, entonces su recorrido tiene una sola dirección posible. Por lo que es muy importante mantener la referencia al nodo inicial, siendo la única manera de poder acceder a los demás nodos.

Doblemente Enlazada: En la lista doblemente enlazada, cada nodo va a tener un puntero a su nodo siguiente y al nodo anterior, exceptuando el caso del nodo inicial y final, que no van a tener referencia a nodo anterior y nodo siguiente respectivamente.

Lista Circular: En la lista circular, ocurre como en la lista simplemente enlazada, cada nodo apunta a su siguiente, pero con el cambio de que el elemento final no apunta a NULL, sino que apunta al nodo inicial.

Lista Circula Doblemente Enlazada: La lista circular doblemente enlazada, es como una lista circular, con la adición de que cada nodo no solo apunta a su siguiente sino que también apunta a su anterior. Además nodo final sigue apuntando a nodo inicial, con el agregado de que nodo inicial apunta también al nodo final.

En las listas se puede operar con cada nodo sin restricción de su posición.

Conjunto mínimo de operaciones:

- Crear() -> Crea la lista reservando memoria.
- Destruir() -> Destruye la lista y libera la memoria que ocupa.
- Insertar() -> Agrega un elemento a la lista, siempre por el final o en una posición específica.
- Borrar() -> Elimina un elemento de la lista, siempre por el final o en una posición específica.
- Primero() -> Permite saber cual es el elemento al frente de la cola.
- Ver_elemento() -> Permite saber cual es el elemento actual, o el elemento en alguna posición específica.
- Esta_Vacia() -> Permite saber si la lista tiene elementos o no.
- Destruir() -> Destruye la lista y libera la memoria que ocupa.

COLA

Una cola se implementa a partir de nodos enlazados, cada uno con su siguiente. La idea principal de una cola es que cada elemento se agrega usando una función "encolar()", que lo que hace es agregarlo siempre por el nodo final de esta. Y para extraer un elemento se usa una función "desencolar()", que elimina el elemento en el nodo inicial siempre. Se tiene siempre el puntero al nodo inicial para poder recorrerlo. Conjunto mínimo de operaciones:

- Crear() -> Crea la cola reservando memoria.
- Destruir() -> Destruye la cola y libera la memoria que ocupa.
- Encolar() -> Agrega un elemento a la cola, siempre por el final.
- Desencolar() -> Elimina un elemento a la cola, siempre por el frente.
- Primero() -> Permite saber cual es el elemento al frente de la cola.
- Esta_Vacia() -> Permite saber si la cola tiene elementos o no.

PILA

La idea de un TDA Pila, es la de usando nodos enlazados, agregar y eliminar elementos siempre por el mismo lugar: el nodo final. Conjunto mínimo de operaciones:

- Crear() -> Crea la pila reservando memoria.
- Apilar() -> Agrega un elemento a la pila, siempre por el final, y aumenta el tope.
- Desapilar() -> Elimina un elemento de la pila, siempre por el final, y disminuye el tope
- Esta_Vacia() -> Permite saber si la pila tiene elementos o no.
- Obtener_Tope() -> Devuelve la cantidad de elementos que tiene la pila.
- Destruir() -> Destruye la pila y libera la memoria que ocupa.

2. Objetivo de la actividad y Detalles de Implementación

El objetivo de esta actividad era el de implementar un TDA Lista a partir de nodos simplemente enlazados, y a partir de este, poder reutilizar lo hecho para implementar un TDA Pila y un TDA Cola. La implementación del TDA Pila y el TDA Cola se hizo utilizando las funciones convenientes antes implementadas en Lista, y casteando los respectivos TDA buscados.

1. Detalles de algunas funciones de LISTA:

lista_t* lista_crear() : Reserva un bloque de memoria en el heap de tantos bytes como tenga lista_t, para lista, lo inicializa con ceros, y devuelve un puntero al inicio del bloque de memoria reservado. En caso de error devuelve NULL.

nodo_t* crear_nodo() : Reserva un bloque de memoria en el heap de tantos bytes como tenga nodo_t, para un nodo, lo inicializa con ceros, y devuelve un puntero al inicio del bloque de memoria reservado. En caso de error devuelve NULL.

nodo_t* cargar_nodo(nodo_t* nodo_a_cargar, void* elemento) : Recibe un puntero a un nodo NULL de la lista. También recibe un elemento válido. Le asigna memoria que fue reservada con crear_nodo() al puntero nodo recibido, y asigna el elemento recibido al nodo creado. Devuelve puntero a el nodo cargado. En caso de error al reservar memoria, devuelve NULL.

lista_t* lista_insertar(lista_t* lista, void* elemento) : Asigna al final de lista un nuevo nodo, con el elemento recibido. Devuelve NULL si no pudo insertar el elemento a causa de un error, o la lista en caso de éxito. Puede darse el caso de que la lista esté vacía, entonces se asigna el primer nodo cargado a nodo_inicio, y también se apunta nodo_fin hacia él, porque al solo haber un elemento, el inicio y el fin serán lo mismo. En cambio, si ya hay por lo menos un elemento, se va a cargar el nuevo nodo siempre en nodo_fin->siguiente, porque si se cargase directamente en nodo_fin estaría asignandole al puntero una nueva dirección de memoria, perdiendo así la dirección de memoria que había antes asignada. Luego, hago que nodo_fin apunte a nodo_fin->siguiente, y a nodo_fin->siguiente le asigno NULL, para marcar correctamente el nuevo fin de la lista. Cada vez que se cargue un elemento aumento lista->cantidad y devuelvo la lista cargada.

Diagrama 1.

lista_t* lista_insertar_en_posicion(lista_t* lista, void* elemento, size_t posicion) : Asigna en la posicion(a partir de 0) indicada de lista un nuevo nodo, con el elemento recibido. En caso de no existir la posicion indicada, lo inserta al final. Devuelve NULL si no pudo insertar el elemento a causa de un error, o la lista en caso de exito. Si la posición es mayor o igual a la cantidad de elementos que hay en la lista, sé que siempre se tendrá que agregar el nuevo nodo al final de la lista, por eso llamo a lista_insertar() que hace eso. Cuando la posición a la que le tengo que asignar el nuevo nodo es diferente a la del final y la del inicio, me guardo la referencia al nodo anterior al de la posición buscada. Luego, le asigno a nodo_a_insertar->siguiente al nodo que desplazó. Esto porque al nodo anterior al nodo_a_insertar, tengo que cargarle su siguiente como el mismo nodo_a_insertar(nodo_anterior->siguiente = nodo_anterior), y si hubiese hecho esto primero, hubiese perdido la referencia al nodo desplazado y así a todos sus nodos siguientes. Si la posición a insertar es cero, solo tengo que hacer que el nodo_a_insertar->siguiente apunte a lo que es el nodo_inicio, y una vez guarda la referencia a los demás nodos, actualizo nodo_inicio como el nodo_a_insertar.

Diagrama 2.

void* lista_quitar(lista_t* lista) : Quita de la lista el elemento que se encuentra en la ultima posición y libera la memoria que ocupa. Devuelve el elemento removido de la lista o NULL en caso de error.

En este función lo que hago es, primero guardarme el elemento de la posición final, que se va a eliminar, en un auxiliar. Esto porque luego voy a liberar el bloque de memoria en donde está ese elemento, y si no hiciera esto, lo perdería. Luego distingo entre el caso en que haya un solo elemento en la lista, y el caso en el que haya más de un elemento.

En el primero, libero la memoria reservada del nodo_inicio, que era la única que había. Y le asigno NULL a nodo_inicio y a nodo_fin para saber que ya no tengo elementos y no debo acceder a esos punteros. En el segundo caso, recorro toda la lista partiendo desde el nodo_inicio(es mí único punto de partida), hasta llegar hacia el nodo anterior al nodo_fin. Libero la memoria perteneciente a nodo_fin, y después le asigno a nodo_fin el nodo anterior que me había guardado, también cargo nodo_fin->siguiente como NULL para saber que no debo acceder ahí. En ambos casos resto un elemento a lista->cantidad y devuelvo lista.

void* lista_quitar_de_posicion(lista_t* lista, size_t posicion) : Quita de la lista el elemento que se encuentra en la posición(a partir de 0) indicada, y libera le memoria que ocupa. En caso de no existir esa posición se intentará borrar el último elemento. Devuelve el elemento removido de la lista o NULL en caso de error. En la función se distinguen tres casos: Posición a borrar es la del nodo_fin o mayor, posición a borrar es la de nodo_inicio, o posición a borrar no entre en ninguno de los anteriores casos.

En el primero de los casos, se debe hacer lo que hace lista_quitar(), por eso se la invoca. En el segundo, me guardo la dirección de memoria a liberar en un auxiliar, para no perder la referencia. Si hay un solo elemento, libero la memoria y referencio nodo_inicio y nodo_fin como NULL. Si hubiese más de un elemento, asigno a nodo_inicio su siguiente(nodo_inicio->siguiente). Y libero la memoria con el auxiliar que tenía. En el tercer caso, recorro la lista desde nodo_inicio hasta un el nodo anterior al nodo en la posición a eliminar, me guardo la posición a eliminar(nodo_anterior->siguiente). Luego apunto nodo_anterior->siguiente hacia nodo_a_eliminar->siguiente, de esta manera me "salteo" el nodo que voy a eliminar, y queda todo enlazado correctamente. Por último, libero la memoria de nodo_a_eliminar.

Diagrama 3.

lista_destruir(lista_t* lista) : Libera la lista y toda la memoria utilizada por la misma. Libera primero cada nodo, y luego libera la lista.

bool lista_iterador_tiene_siguiente(lista_iterador_t* iterador) : Devuelve true siempre que iterador->corriente no sea NULL, esto porque, por ejemplo, en una iteración con un "for", siempre que iterador->corriente sea válido significará que habrá un elemento que no ha entrado en la iteración.

bool lista_iterador_avanzar(lista_iterador_t* iterador) : Devuelve true siempre que iterador->corriente->siguiente sea no NULL, o sea, que el corriente no sea el nodo_fin. En el caso de que sea el nodo_fin, se apunta iterador->corriente a NULL para saber que terminó de iterar y que lista_iterador_tiene_siguiente() corte correctamente la iteración.

size_t lista_con_cada_elemento(lista_t* lista, bool (*funcion) (void * **, void** *), **void *contexto) :** Crea un iterador interno con el que recorrer la lista y aplicarle la función a cada elemento, la función devuelve true si se debe seguir iterando con los demás elementos o false en caso de que se deba cortar la iteración. Una vez terminada la iteración, se libera la memoria ocupada por el iterador y se devuelve el numero de elementos iterados.

2. Detalles de implementación de PILA a partir de "lista.h":

pila_t* pila_crear() : Utiliza lista_crear() y castea su retorno en pila_t* para poder devolverlo bien. O sea, estaría reservando memoria del tamaño de lista_t.

pila_t* pila_apilar(pila_t* pila, void* elemento) : Como en una pila siempre se apila por el tope, puedo llamar a lista_insertar() que siempre agrega los nuevos nodos al final de la lista. Una vez apilado el nuevo elemento, devuelve la pila.

void* pila_desapilar(pila_t* pila) : Desapila un elemento y lo devuelve. Como sé que en una pila siempe se desapila por el tope, llamo a lista_quitar_de_posicion() y le paso como posición a eliminar la del último nodo, que sería la cantidad total de elementos.

Diagrama 4.

3. Detalles de implementación de COLA a partir de "lista.h":

cola_t* cola_crear() : Similar al procedimiento de pila_crear(). Utiliza lista_crear() y castea su retorno en cola_t* para poder devolverlo bien. O sea, estaría reservando memoria del tamaño de lista_t.

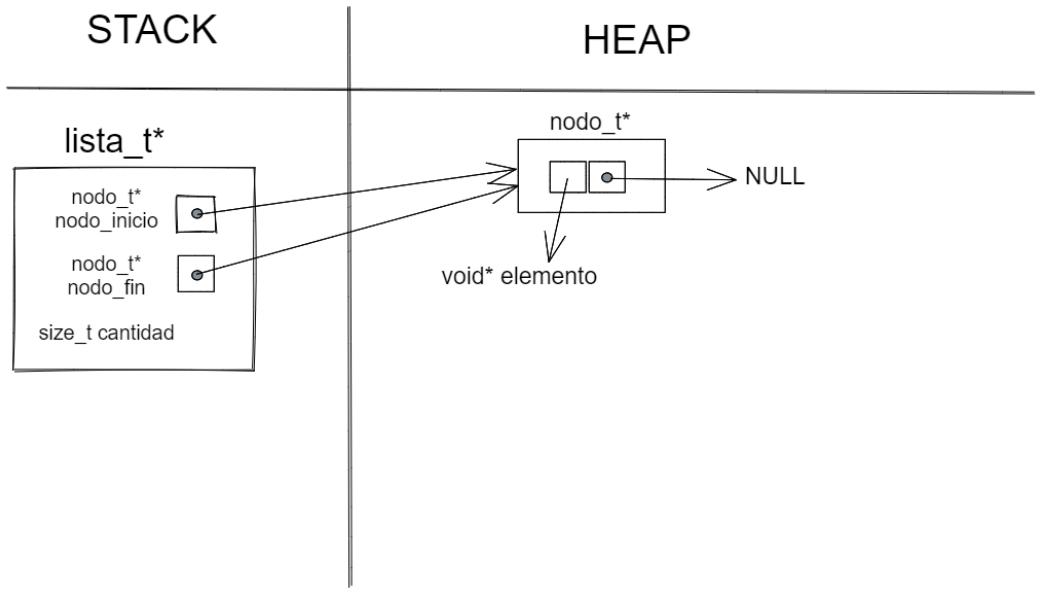
cola_t* cola_encolar(cola_t* cola, void* elemento) : Sabiendo que en la cola, para agregar encolar siempre se hace por el final de esta, llamo a lista_insertar() y casteo su retorno como cola_t*. Devuelvo la cola con su nuevo nodo encolado.

void* cola_desencolar(cola_t* cola) : Como para desencolar siempre se empieza por el elemento del frente, llamo a lista_quitar_de_posicion() y le envío como posición, la inicial que es cero. Devuelve el elemento desencolado.

Diagrama 5.

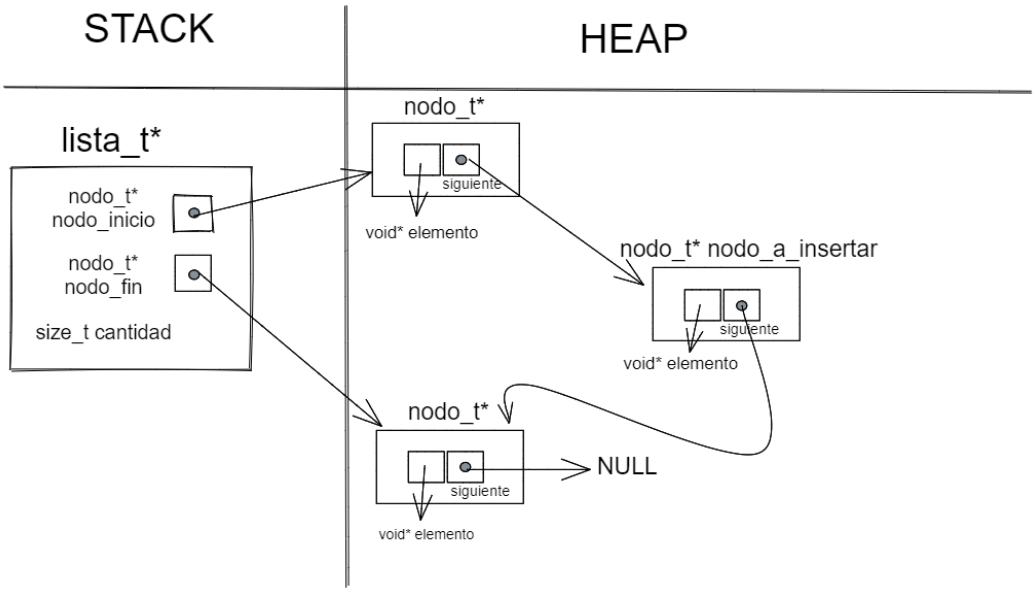
3. Diagramas

1. Diagrama 1



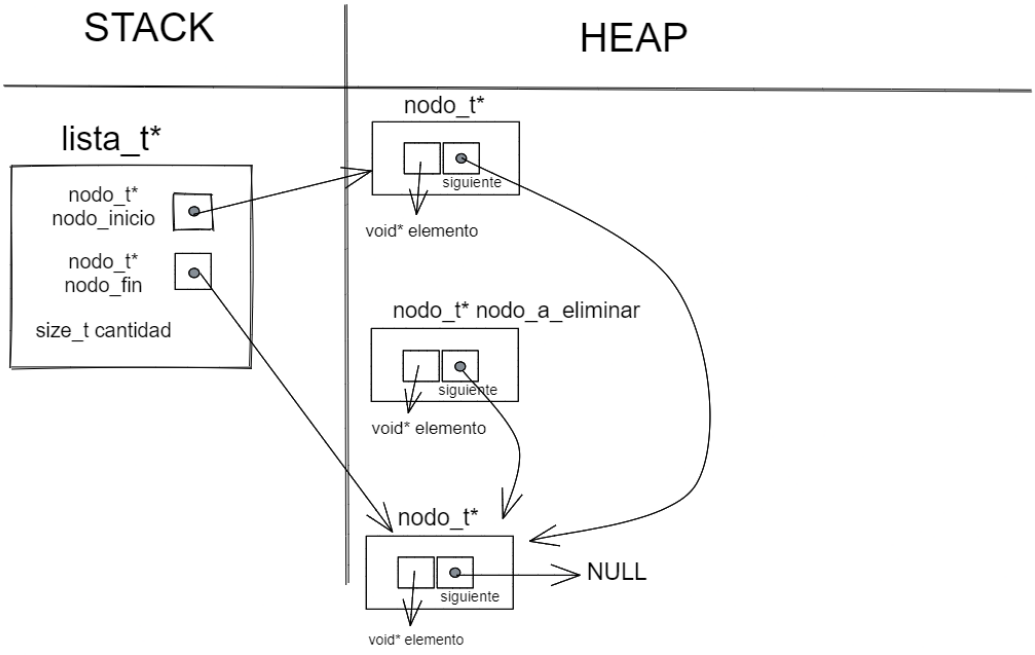
Ejemplo de como quedaría una lista con un solo elemento.

2. Diagrama 2



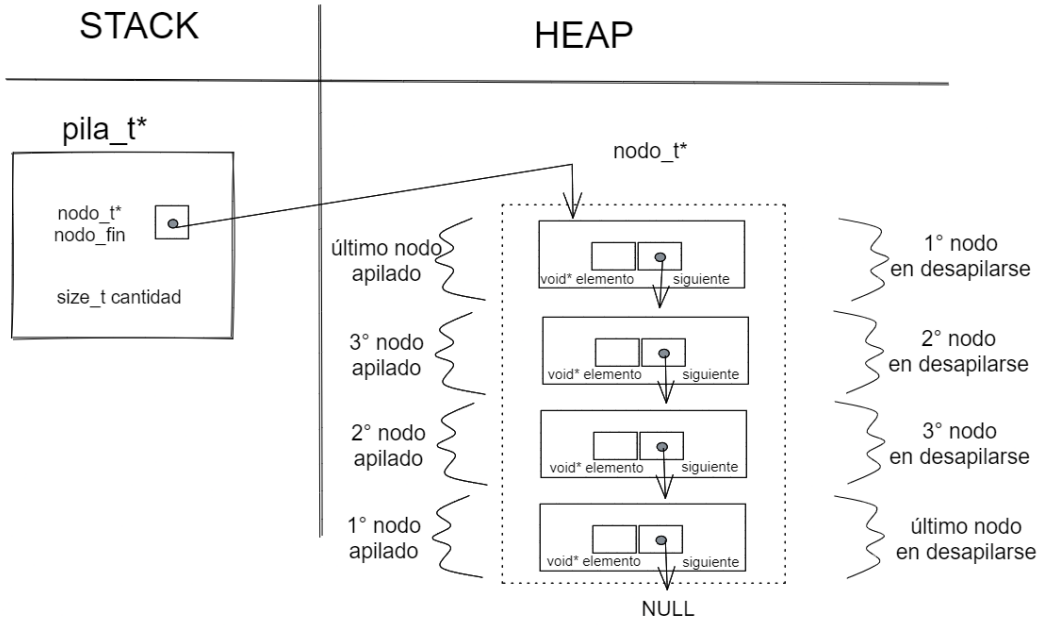
Ejemplo de como quedaría el insertar un elemento en una posición de en medio de una lista(en este caso de tres elementos).

3. Diagrama 3



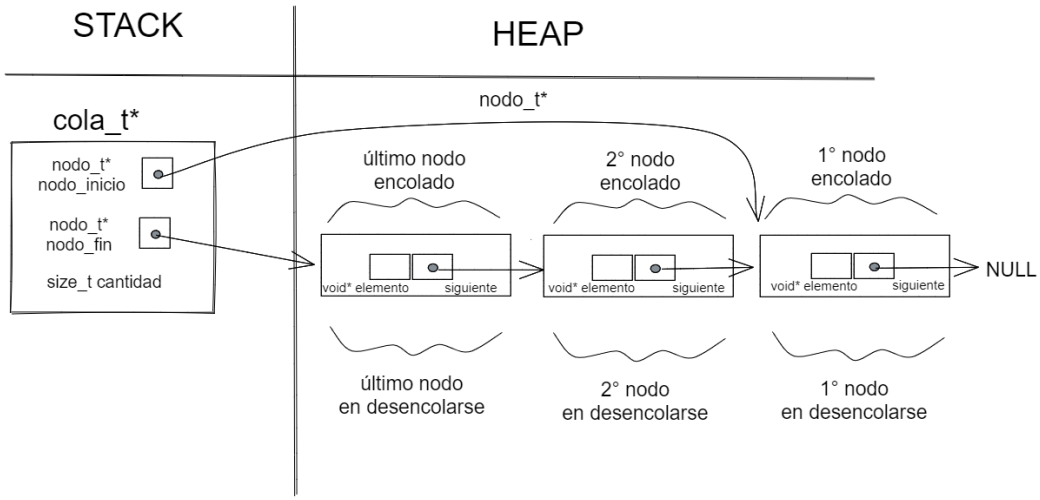
Ejemplo de como quedaría un elemento de una posición en medio de la lista, listo para eliminar, se aprecia que no hay ninguna manera de llegar a él siguiendo la lista.

4. Diagrama 4



Ejemplo de como quedaría una pila de cuatro elementos ya cargada, señalizando el orden en que se apilaron y el orden en que deberían desapilarse. El único puntero con el que se puede acceder a la pila apunta al último nodo creado. Aclaración: En este trabajo realizado pila_t* tendría un nodo_inicio, pero como no se debería operar con él, no está agregado al diagrama.

4. Diagrama 5



Ejemplo de como quedaría una cola de tres elementos ya cargada, señalizando el orden en que se encolaron y el orden en que deberían desencolarse.